NYCU DLP 2025 Lab1
Backpropagation

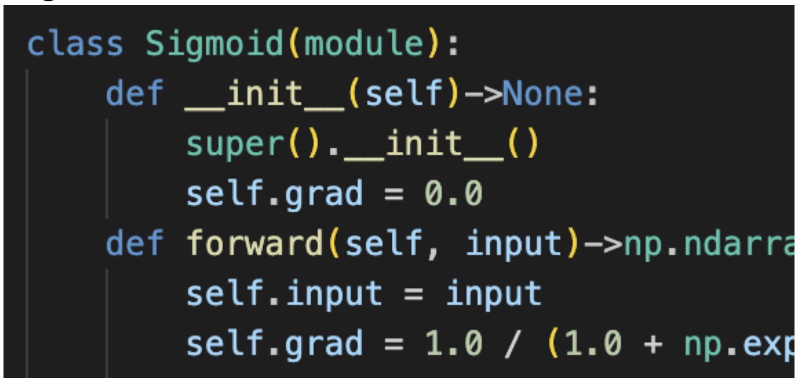
110550110 林書愷

1. Introduction:

In this lab, I implemented a neural network with three hidden layers and performed both forward and backpropagation. I conducted multiple experiments by varying parameters such as the learning rate, using different optimizers, and testing different activation functions to analyze their impact on the neural network during training.

2. Impementation:

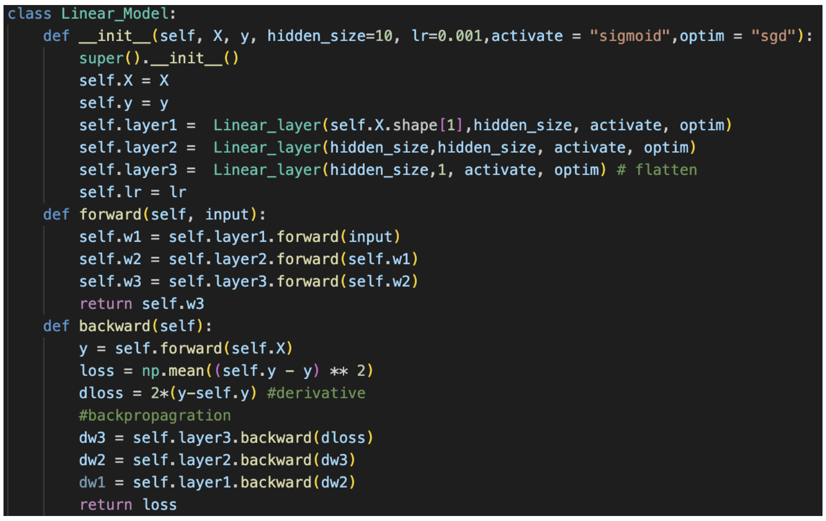
a. Sigmoid function:



```python
class Sigmoid(module):
    def __init__(self)->None:
        super().__init__()
        self.grad = 0.0
    def forward(self, input)->np.ndarra
        self.input = input
        self.grad = 1.0 / (1.0 + np.exp
```

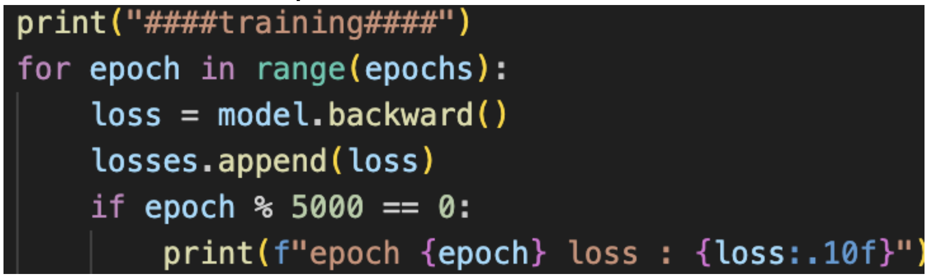
In my implementation of sigmoid, I separated into two parts. For the first part, it implemented the original sigmoid function, σ(x)=1/(1 + e^-x) and output the gradient. The second part, it implemented the the derivation of sigmoid, σ′(x)=σ(x)(1−σ(x)) and output the adjusted gradient weight that possess minimum loss.

b. Neural Network architecture:



```python
class Linear_Model:
    def __init__(self, X, y, hidden_size=10, lr=0.001,activate = "sigmoid",optim = "sgd"):
        super().__init__()
        self.X = X
        self.y = y
        self.layer1 =  Linear_layer(self.X.shape[1],hidden_size, activate, optim)
        self.layer2 =  Linear_layer(hidden_size,hidden_size, activate, optim)
        self.layer3 =  Linear_layer(hidden_size,1, activate, optim) # flatten
        self.lr = lr
    def forward(self, input):
        self.w1 = self.layer1.forward(input)
        self.w2 = self.layer2.forward(self.w1)
        self.w3 = self.layer3.forward(self.w2)
        return self.w3
    def backward(self):
        y = self.forward(self.X)
        loss = np.mean((self.y - y) ** 2)
        dloss = 2*(y-self.y) #derivative
        #backpropagation
        dw3 = self.layer3.backward(dloss)
        dw2 = self.layer2.backward(dw3)
        dw1 = self.layer1.backward(dw2)
        return loss
```

The neural network architecture is formed with three hidden layers with all followed by activation function and optimizer. The hidden layers are built with layer class including linear layer with no backpropagation, linear layer and convolution layer. Activation function and optimizer can be selected with sigmoid, relu, sgd, adagrad, momentum during initialization session. To train the model, just call backward function. It will fit the model's weight with input data by calling forward function and calculate the loss. After that, it will call each layers backpropagation to minimize the loss and update the layers' weight. Finally it returns the loss of current step.

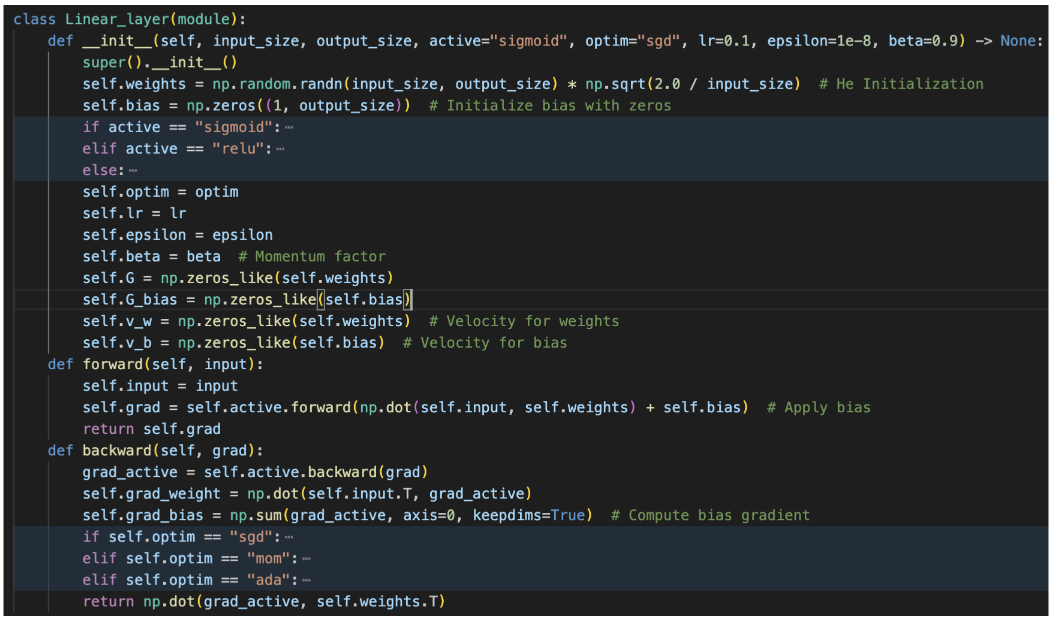


```
print("####training####")
for epoch in range(epochs):
    loss = model.backward()
    losses.append(loss)
    if epoch % 5000 == 0:
        print(f"epoch {epoch} loss : {loss:.10f}"
```

c. Backpropagation:

The backpropagation step I implemented in the layer class. For the initialization stage, it will set up its weights with random array same as input * output size and a zero array as bias. Activation function and optimizer will also be set up based on the pass in variables. The forward function implements the fitting task as it will calculate the linear formula with input times its weight and bias added and then go through activation function.

The backward function is the core of back propagation as will minimize the loss by modifying its weight. It will first go through the derivation of activation function and then update the gradient of weight and bias. Then go through the optimizer to update weight and bias by subtracting the learning and gradient weight and bias. Finally, return the gradient of activation function and times weights.

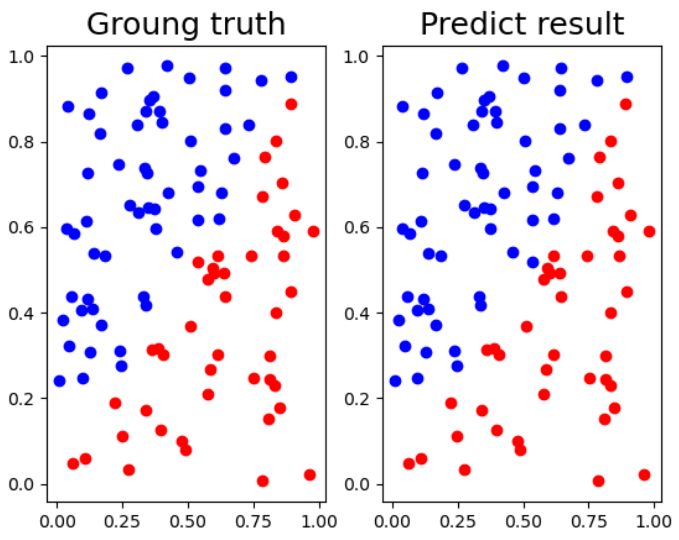


```python
class Linear_layer(module):
    def __init__(self, input_size, output_size, active="sigmoid", optim="sgd", lr=0.1, epsilon=1e-8, beta=0.9) -> None:
        super().__init__()
        self.weights = np.random.randn(input_size, output_size) * np.sqrt(2.0 / input_size)  # He Initialization
        self.bias = np.zeros((1, output_size))  # Initialize bias with zeros
        if active == "sigmoid":⋯
        elif active == "relu":⋯
        else:⋯
        self.optim = optim
        self.lr = lr
        self.epsilon = epsilon
        self.beta = beta  # Momentum factor
        self.G = np.zeros_like(self.weights)
        self.G_bias = np.zeros_like(self.bias)
        self.v_w = np.zeros_like(self.weights)  # Velocity for weights
        self.v_b = np.zeros_like(self.bias)  # Velocity for bias
    def forward(self, input):
        self.input = input
        self.grad = self.active.forward(np.dot(self.input, self.weights) + self.bias)  # Apply bias
        return self.grad
    def backward(self, grad):
        grad_active = self.active.backward(grad)
        self.grad_weight = np.dot(self.input.T, grad_active)
        self.grad_bias = np.sum(grad_active, axis=0, keepdims=True)  # Compute bias gradient
        if self.optim == "sgd":⋯
        elif self.optim == "mom":⋯
        elif self.optim == "ada":⋯
        return np.dot(grad_active, self.weights.T)
```

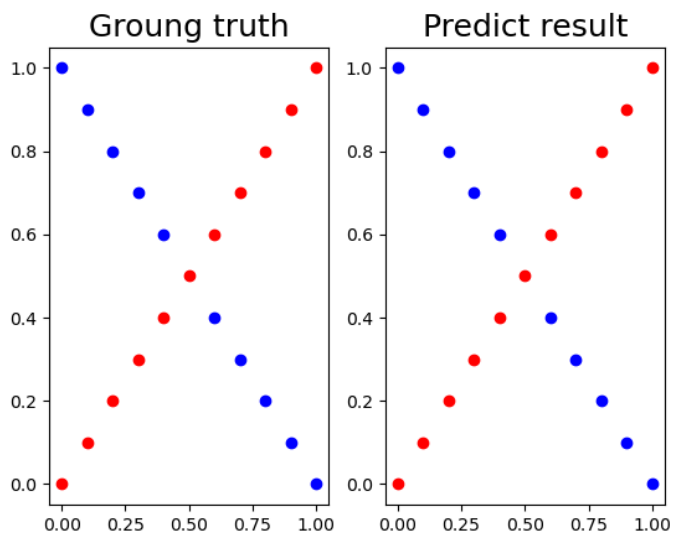
3. Experiment:

a. Screenshot and comparison figure

i. Linear:

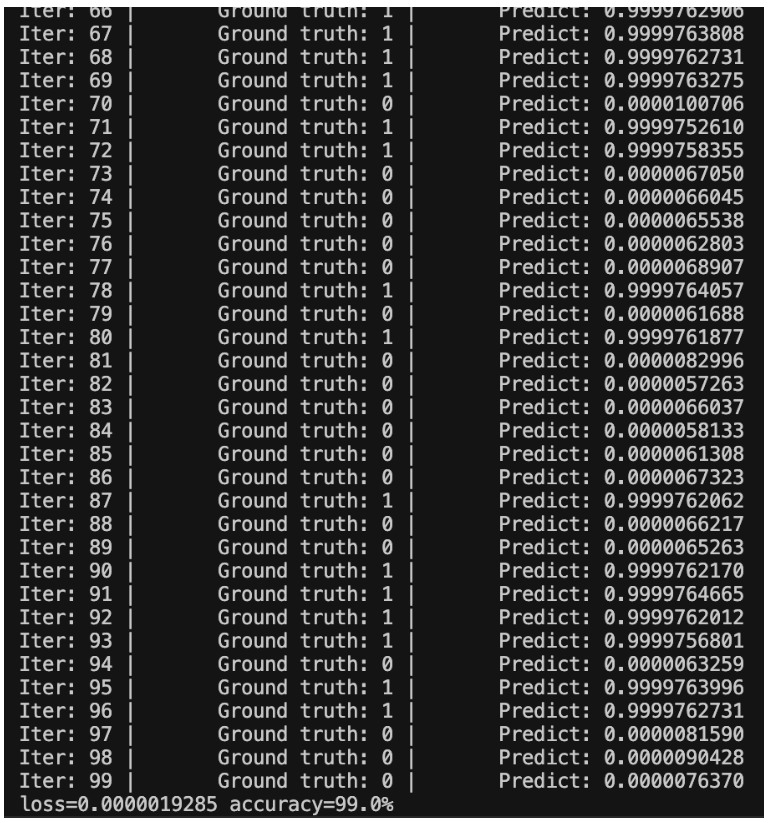


ii. XOR:



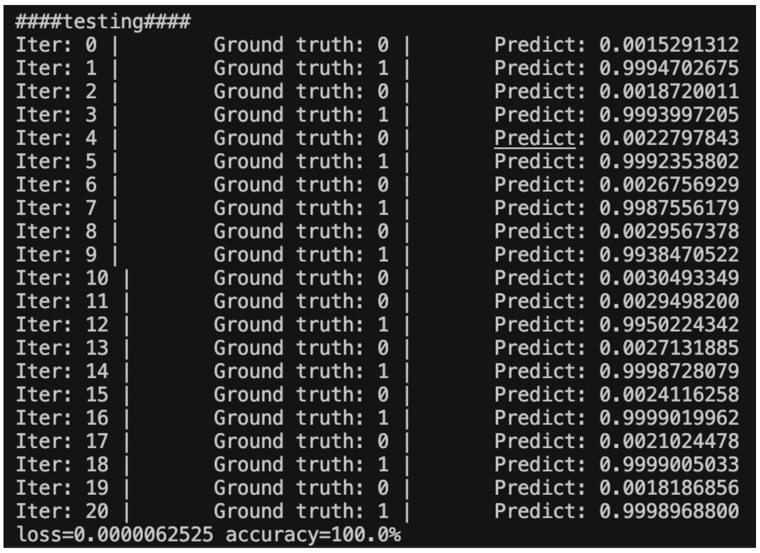
b. Show the accuracy of your prediction

i. Linear:



```
Iter: 66 |        Ground truth: 1 |        Predict: 0.9999762906
Iter: 67 |        Ground truth: 1 |        Predict: 0.9999763808
Iter: 68 |        Ground truth: 1 |        Predict: 0.9999762731
Iter: 69 |        Ground truth: 1 |        Predict: 0.9999763275
Iter: 70 |        Ground truth: 0 |        Predict: 0.0000100706
Iter: 71 |        Ground truth: 1 |        Predict: 0.9999752610
Iter: 72 |        Ground truth: 1 |        Predict: 0.9999758355
Iter: 73 |        Ground truth: 0 |        Predict: 0.0000067050
Iter: 74 |        Ground truth: 0 |        Predict: 0.0000066045
Iter: 75 |        Ground truth: 0 |        Predict: 0.0000065538
Iter: 76 |        Ground truth: 0 |        Predict: 0.0000062803
Iter: 77 |        Ground truth: 0 |        Predict: 0.0000068907
Iter: 78 |        Ground truth: 1 |        Predict: 0.9999764057
Iter: 79 |        Ground truth: 0 |        Predict: 0.0000061688
Iter: 80 |        Ground truth: 1 |        Predict: 0.9999761877
Iter: 81 |        Ground truth: 0 |        Predict: 0.0000082996
Iter: 82 |        Ground truth: 0 |        Predict: 0.0000057263
Iter: 83 |        Ground truth: 0 |        Predict: 0.0000066037
Iter: 84 |        Ground truth: 0 |        Predict: 0.0000058133
Iter: 85 |        Ground truth: 0 |        Predict: 0.0000061308
Iter: 86 |        Ground truth: 0 |        Predict: 0.0000067323
Iter: 87 |        Ground truth: 1 |        Predict: 0.9999762062
Iter: 88 |        Ground truth: 0 |        Predict: 0.0000066217
Iter: 89 |        Ground truth: 0 |        Predict: 0.0000065263
Iter: 90 |        Ground truth: 1 |        Predict: 0.9999762170
Iter: 91 |        Ground truth: 1 |        Predict: 0.9999764665
Iter: 92 |        Ground truth: 1 |        Predict: 0.9999762012
Iter: 93 |        Ground truth: 1 |        Predict: 0.9999756801
Iter: 94 |        Ground truth: 0 |        Predict: 0.0000063259
Iter: 95 |        Ground truth: 1 |        Predict: 0.9999763996
Iter: 96 |        Ground truth: 1 |        Predict: 0.9999762731
Iter: 97 |        Ground truth: 0 |        Predict: 0.0000081590
Iter: 98 |        Ground truth: 0 |        Predict: 0.0000090428
Iter: 99 |        Ground truth: 0 |        Predict: 0.0000076370
loss=0.0000019285 accuracy=99.0%
```

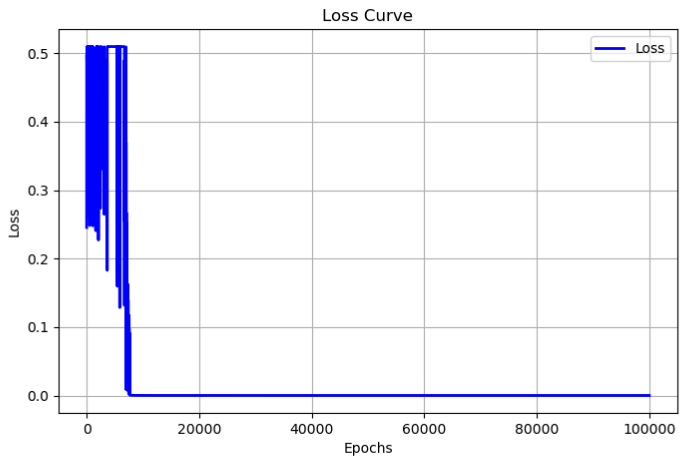
ii. XOR:



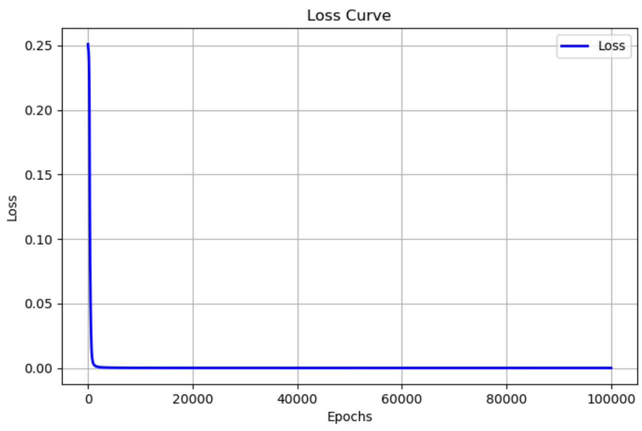
```
####testing####
Iter: 0 |        Ground truth: 0 |       Predict: 0.0015291312
Iter: 1 |        Ground truth: 1 |       Predict: 0.9994702675
Iter: 2 |        Ground truth: 0 |       Predict: 0.0018720011
Iter: 3 |        Ground truth: 1 |       Predict: 0.9993997205
Iter: 4 |        Ground truth: 0 |       Predict: 0.0022797843
Iter: 5 |        Ground truth: 1 |       Predict: 0.9992353802
Iter: 6 |        Ground truth: 0 |       Predict: 0.0026756929
Iter: 7 |        Ground truth: 1 |       Predict: 0.9987556179
Iter: 8 |        Ground truth: 0 |       Predict: 0.0029567378
Iter: 9 |        Ground truth: 1 |       Predict: 0.9938470522
Iter: 10 |       Ground truth: 0 |       Predict: 0.0030493349
Iter: 11 |       Ground truth: 0 |       Predict: 0.0029498200
Iter: 12 |       Ground truth: 1 |       Predict: 0.9950224342
Iter: 13 |       Ground truth: 0 |       Predict: 0.0027131885
Iter: 14 |       Ground truth: 1 |       Predict: 0.9998728079
Iter: 15 |       Ground truth: 0 |       Predict: 0.0024116258
Iter: 16 |       Ground truth: 1 |       Predict: 0.9999019962
Iter: 17 |       Ground truth: 0 |       Predict: 0.0021024478
Iter: 18 |       Ground truth: 1 |       Predict: 0.9999005033
Iter: 19 |       Ground truth: 0 |       Predict: 0.0018186856
Iter: 20 |       Ground truth: 1 |       Predict: 0.9998968800
loss=0.0000062525 accuracy=100.0%
```

c. Learning curve (loss, epoch curve)

i. Linear:



ii. XOR:

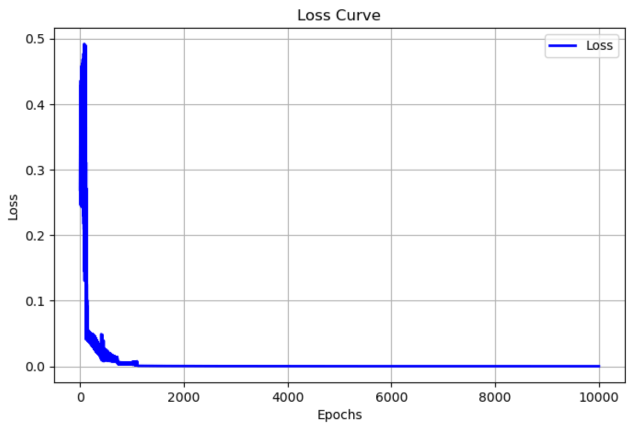


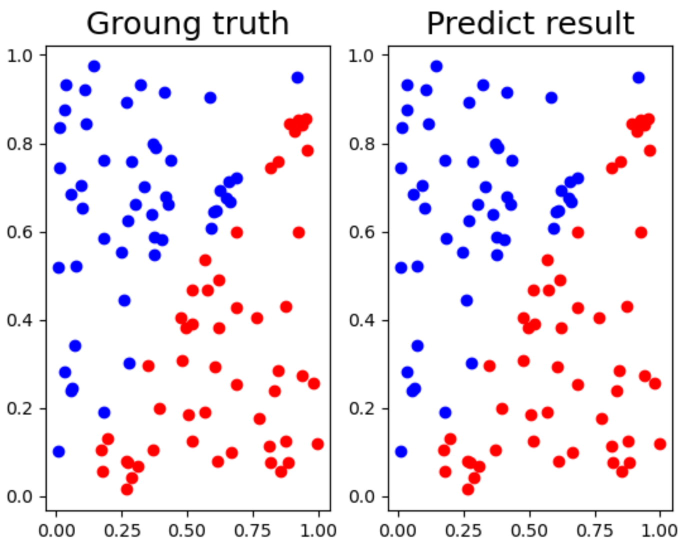
4. Discussion:

a. Different learning rate:

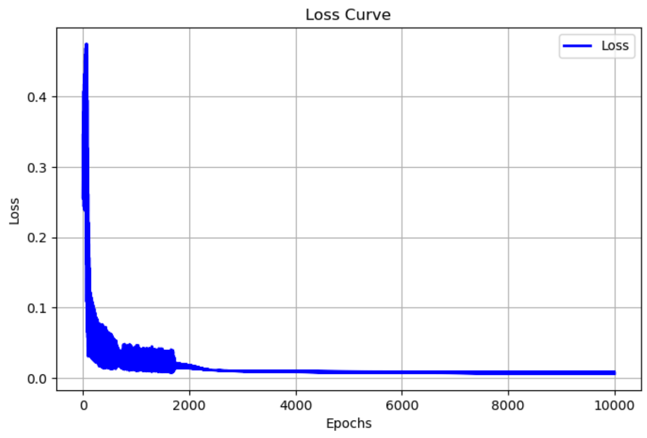
i. Linear:

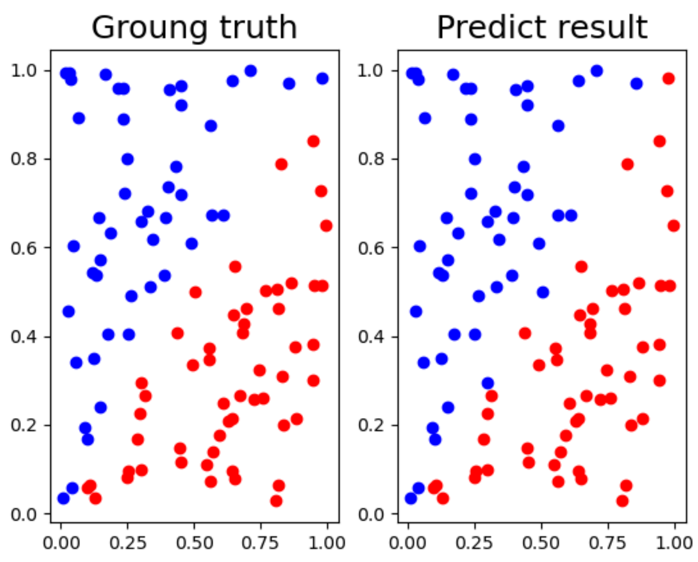
1. lr = 1e-3:





2. lr = 1e-6:

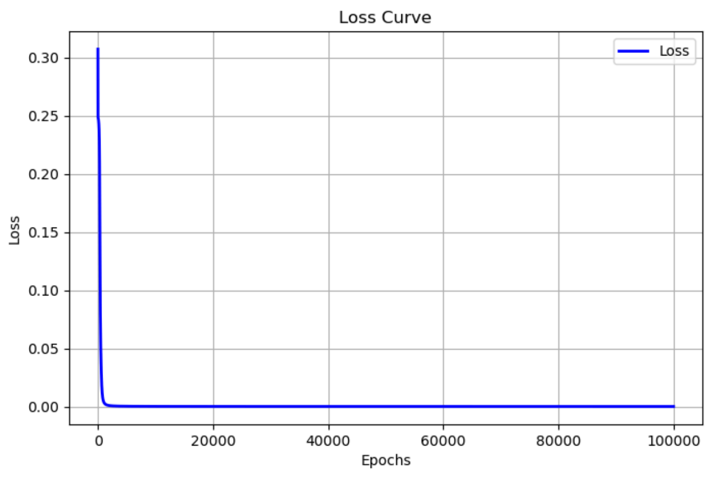


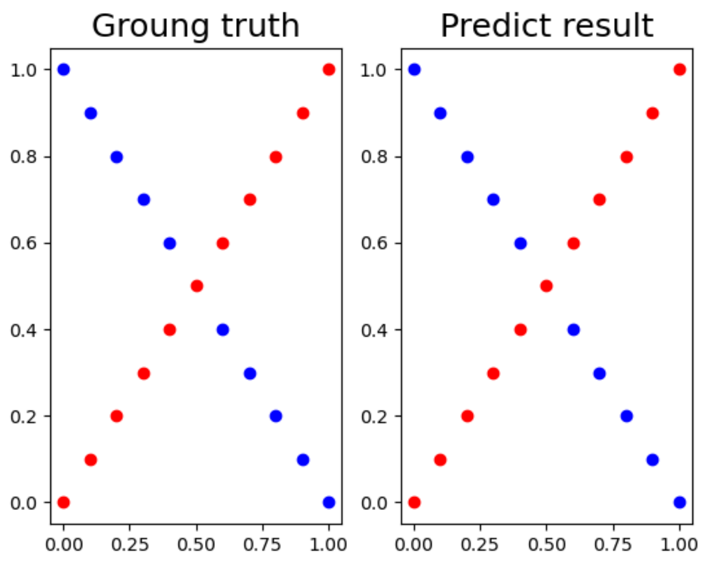


Can discovered that with larger learning rate it will converge faster. Yet in linear dataset, learning rate has minor impact.

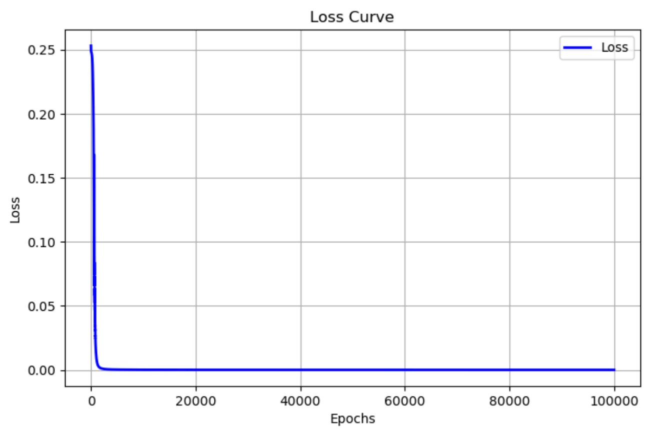
ii. XOR:

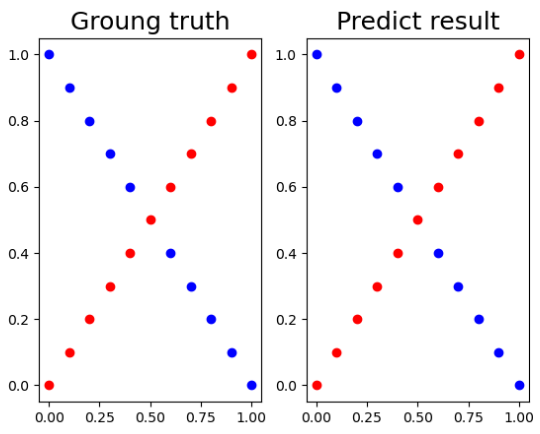
1. learning rate = 1e-3:





2. learning rate = 1e-6:



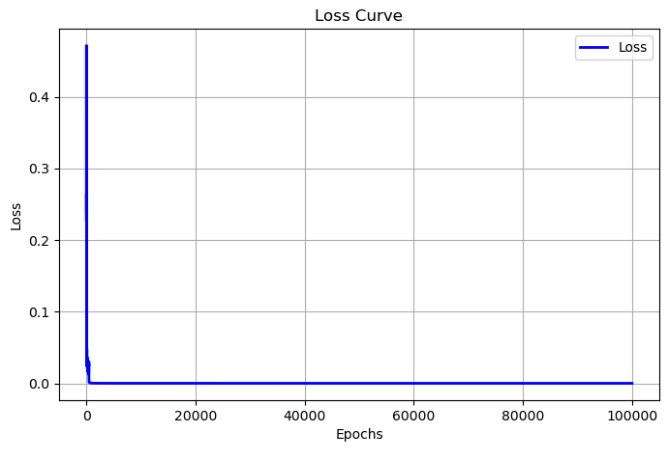


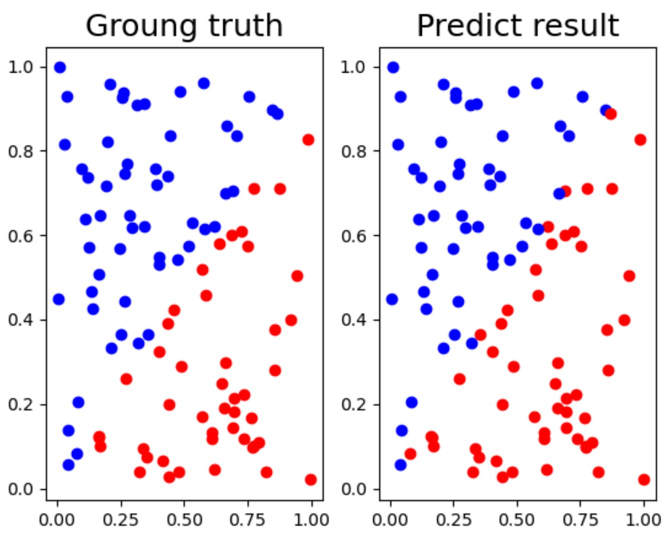
Since XOR dataset is much more easier, the impact of learning rate become less significant in loss. Yet, still can carefully discover that larger lr will converge faster, while accuracy has little impact.

b. Different number of hidden layer unit:

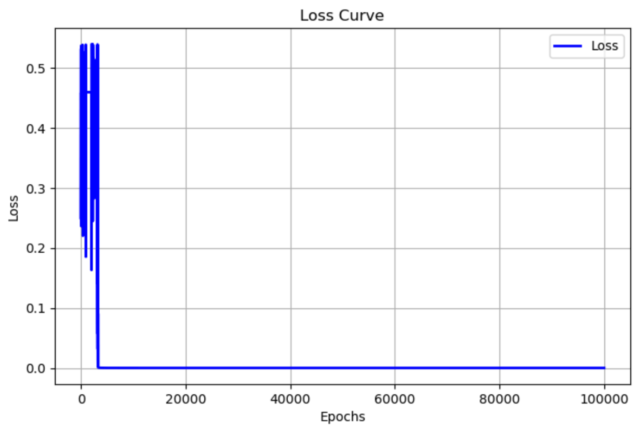
i. Linear:

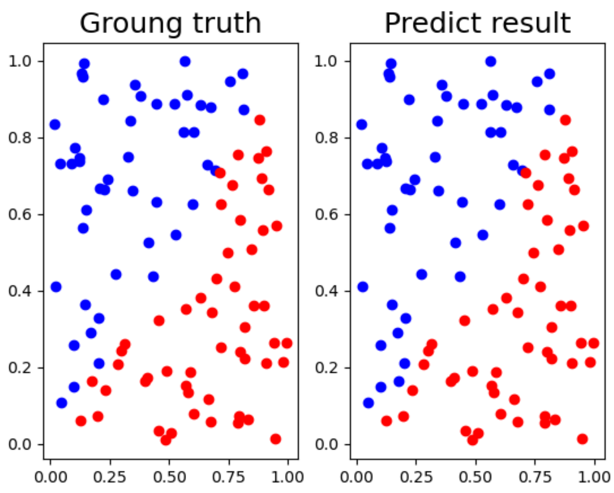
1. hidden unit = 4:





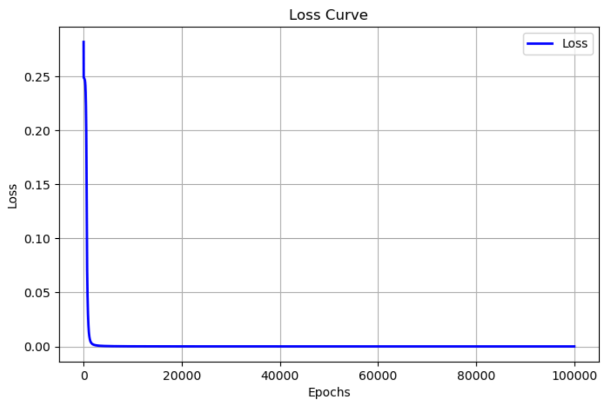
2. hidden unit = 10:

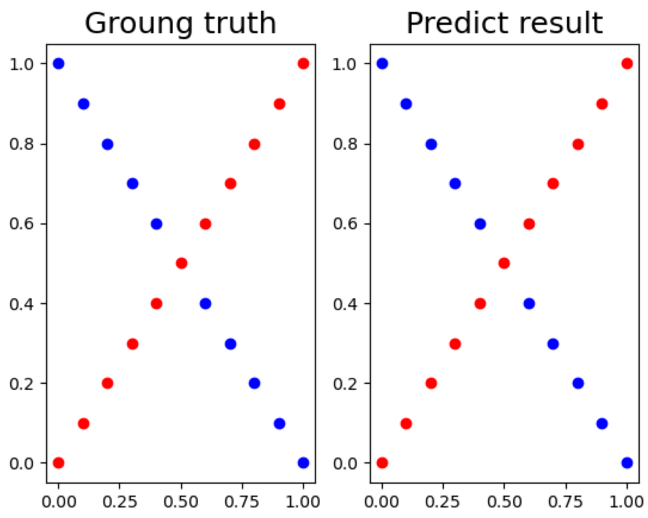




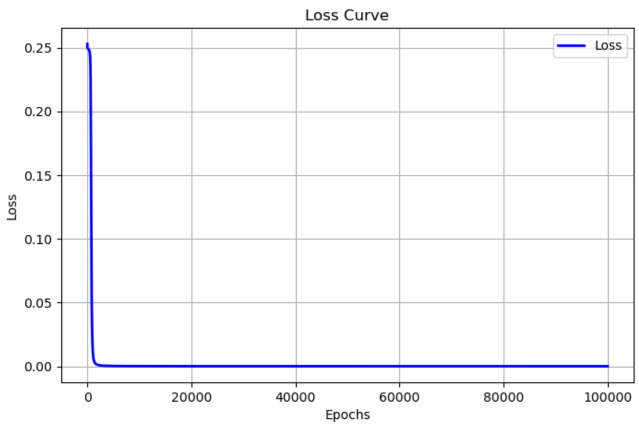
ii. XOR:

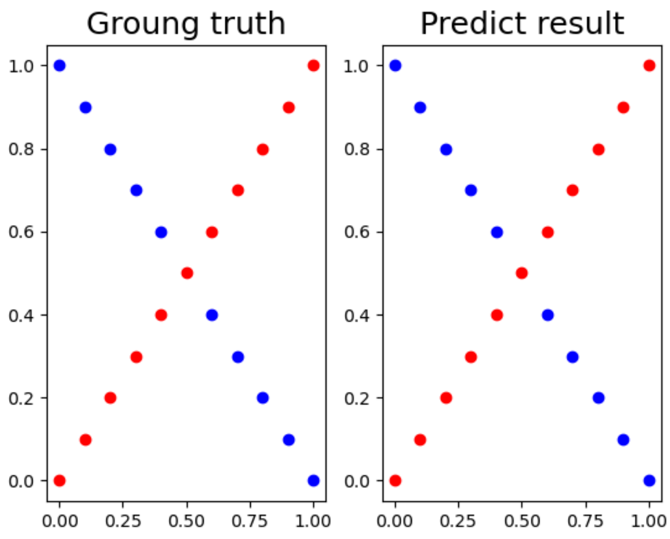
1. hidden unit = 4:





2. hidden unit = 10:

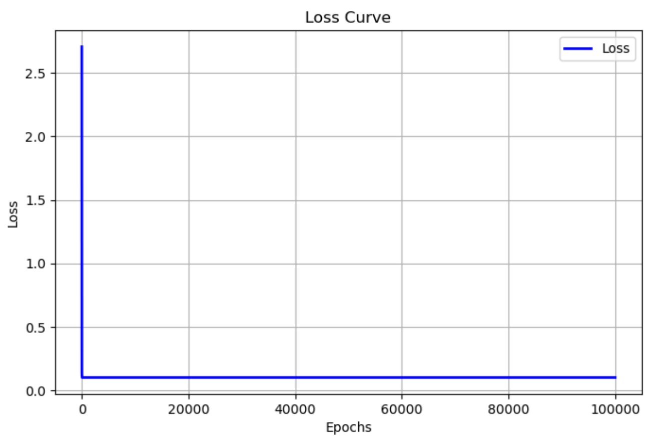


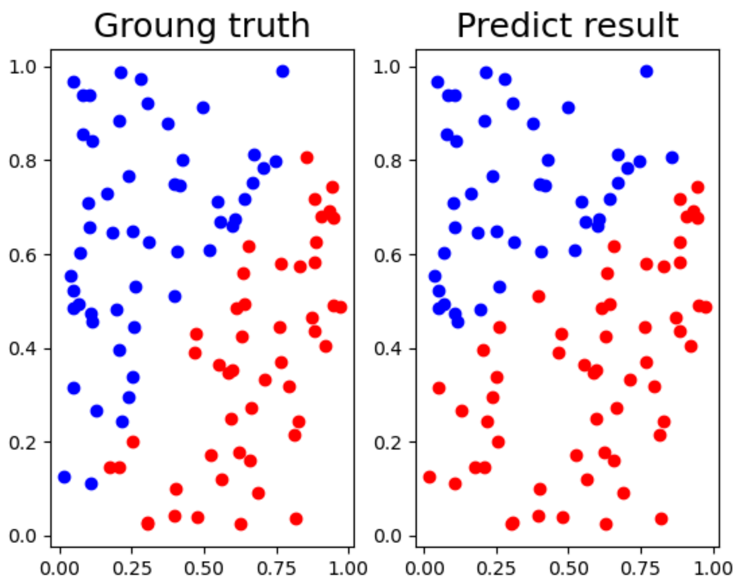


As the figures shown above, the most significant impact of different hidden unit numbers to the performance of the neural network is the convergence of loss. With more units, the neural network becomes more complicated which will cost more time to converge. On the other hand, for more difficult dataset like linear the accuracy will improve with more units as it can memorize more detail of the features.

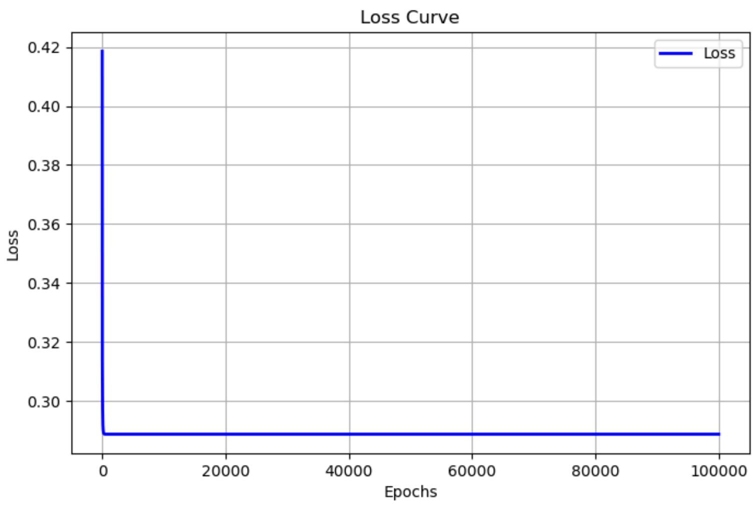
c. Without activation function:

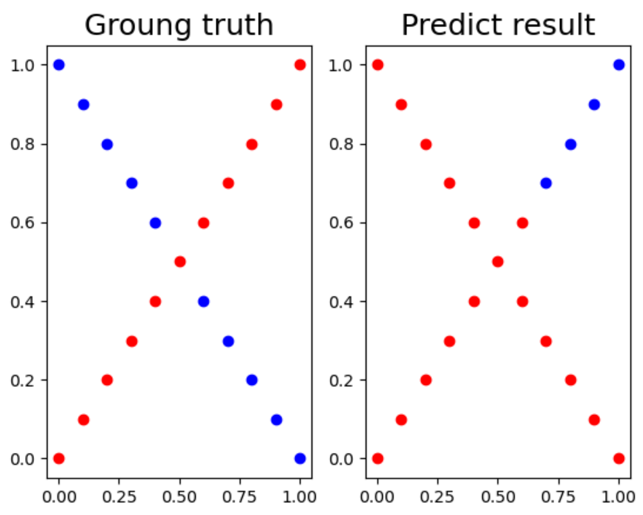
i. Linear:


Loss Curve
Loss
Loss
Epochs

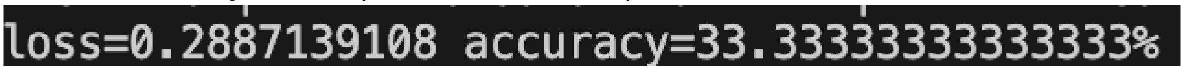

Groung truth
Predict result

ii. XOR:


Loss Curve
Loss
Loss
Epochs



As the above figure shows, in both XOR and linear dataset model without activation function will has a fast convergence but the accuracy is considerably lower.(ie. 33.3% in xor)



```
loss=0.2887139108 accuracy=33.33333333333333%
```

5. Question:

a. Purpose of activation function:

Activation function is applied to the output of each neuron in a neural network and its primary role is to introduce non-linearity to the network and enables the model to learn complex patterns and features. Without activation function, layers of neural network will be just a series of linear transformations which is just same as single layer. Also, activation function can control output range and help stablizing the gradients' update.

b. Learning rate too high or low:

The consequence of learning rate too high will be model weights updated too fast that will cause it unable to converge and eventually lead to an low accuracy situation. On the other hand, learning rate too low will cause model weights updating too slow that it may stuck as local minimum and leads to underfitting.

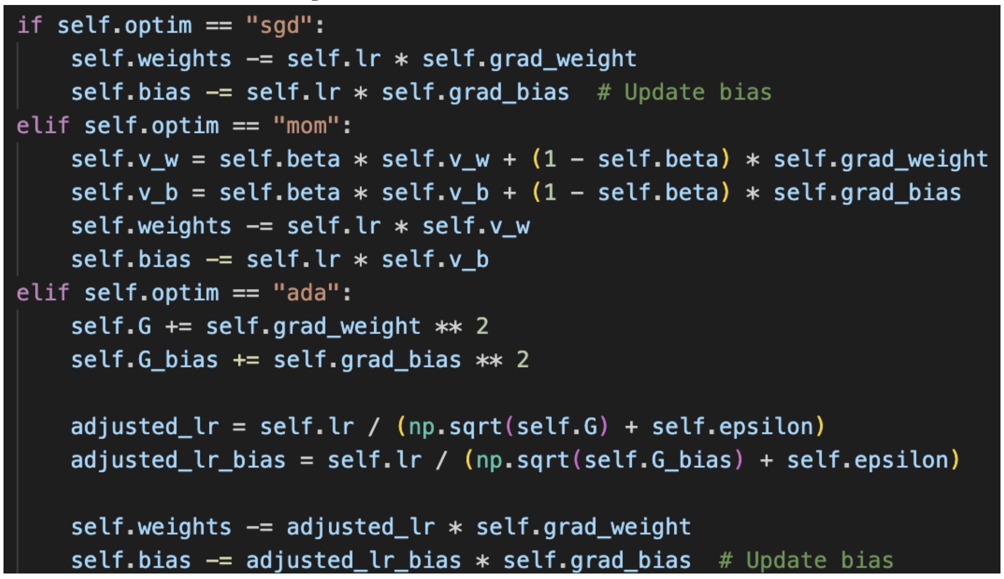
c. Purpose of weight and bias:

For weight, it is the scalar value in the neural network that multiple with input features. It will decide the influence of each features and make the neural network fits the training dataset.

For bias, it is an additional parameter that is added to product of weight and features. Giving an evenly shift effect to neural network which can help avoiding unwanted prediction such as extreme values like zero.(very devastated to deep learning)

6. Extra:

a. Different optimizers:

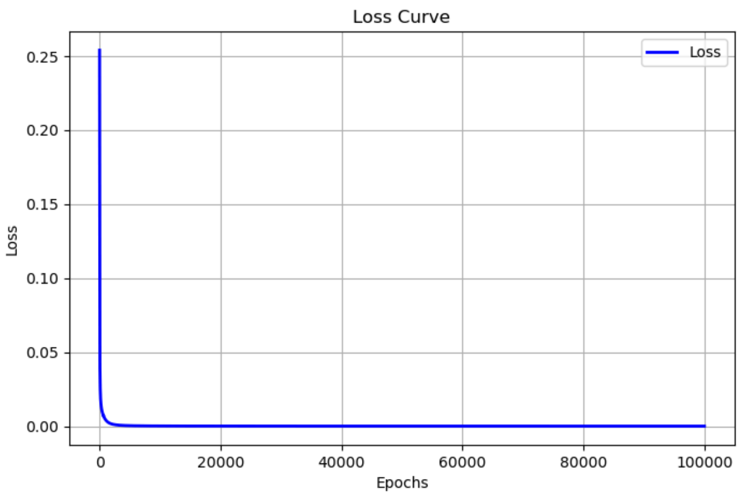


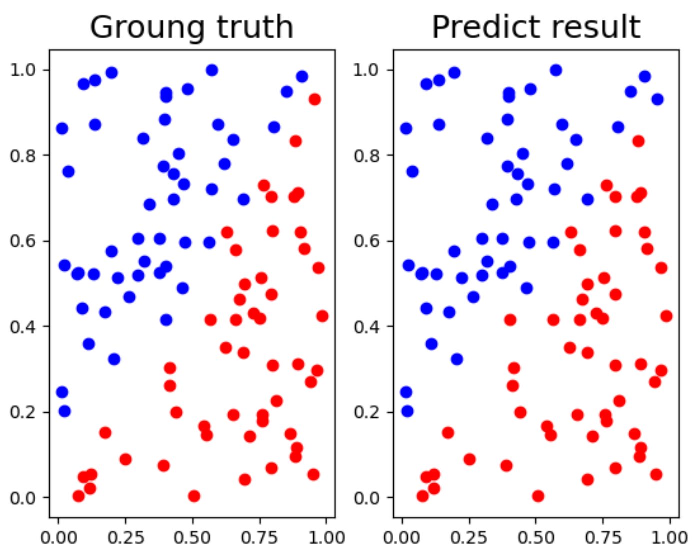
```
if self.optim == "sgd":
    self.weights -= self.lr * self.grad_weight
    self.bias -= self.lr * self.grad_bias  # Update bias
elif self.optim == "mom":
    self.v_w = self.beta * self.v_w + (1 - self.beta) * self.grad_weight
    self.v_b = self.beta * self.v_b + (1 - self.beta) * self.grad_bias
    self.weights -= self.lr * self.v_w
    self.bias -= self.lr * self.v_b
elif self.optim == "ada":
    self.G += self.grad_weight ** 2
    self.G_bias += self.grad_bias ** 2

    adjusted_lr = self.lr / (np.sqrt(self.G) + self.epsilon)
    adjusted_lr_bias = self.lr / (np.sqrt(self.G_bias) + self.epsilon)

    self.weights -= adjusted_lr * self.grad_weight
    self.bias -= adjusted_lr_bias * self.grad_bias  # Update bias
```

I implemented three different optimizers including sgd, momentum, and AdaGrad. Below are comparison with network using same learning rate = 1e-6, hidden unit size = 10, and sigmoid as activation function but with different optimizers:

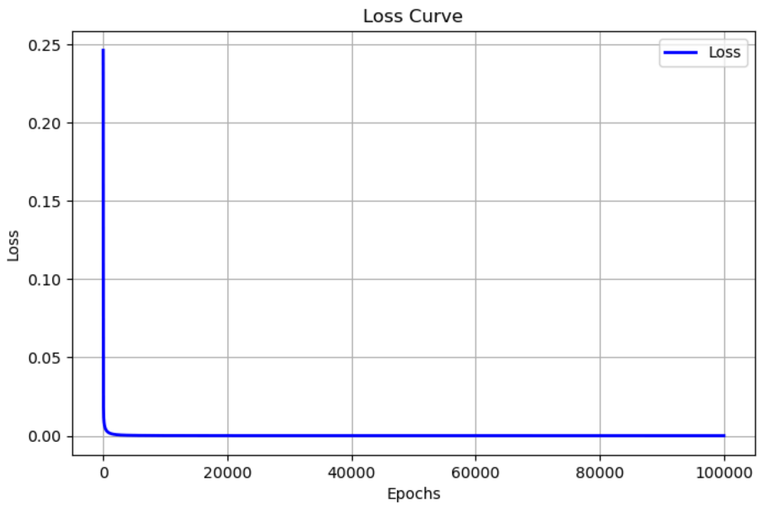
i. Linear:

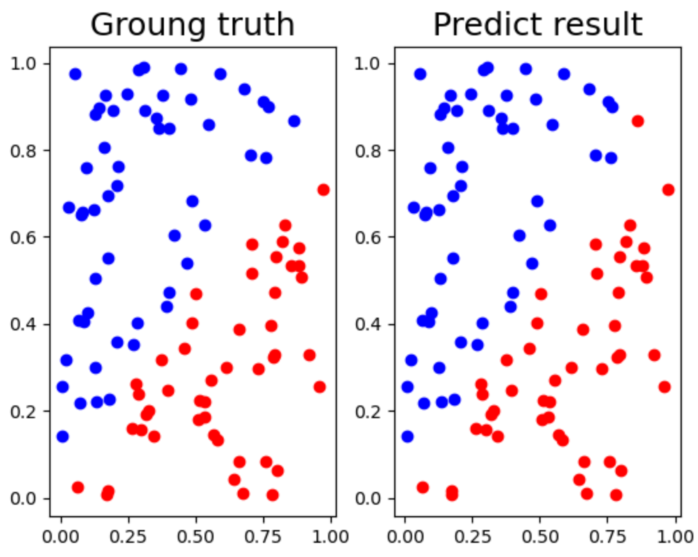
1. AdaGrad:




Groung truth
Predict result

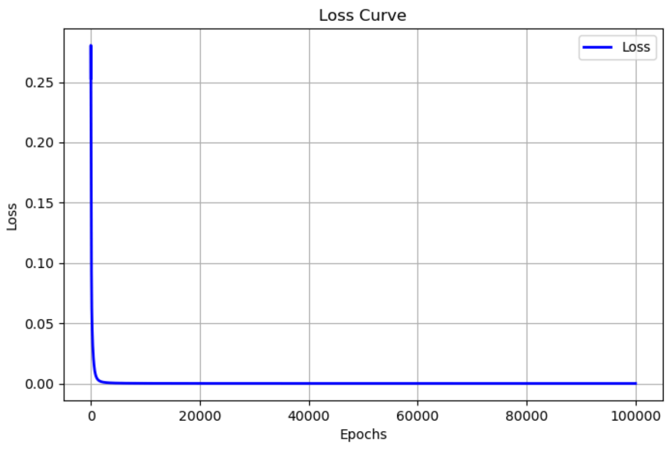
2. Momentum:

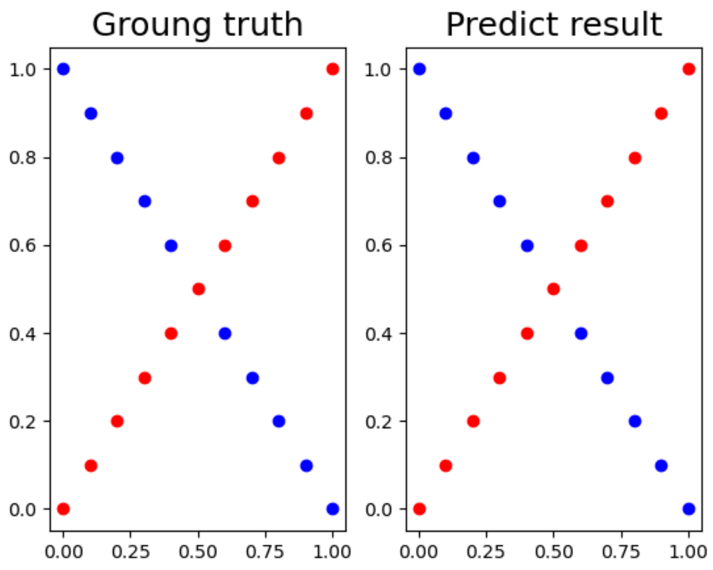

Loss Curve
Loss
Epochs


Groung truth
Predict result

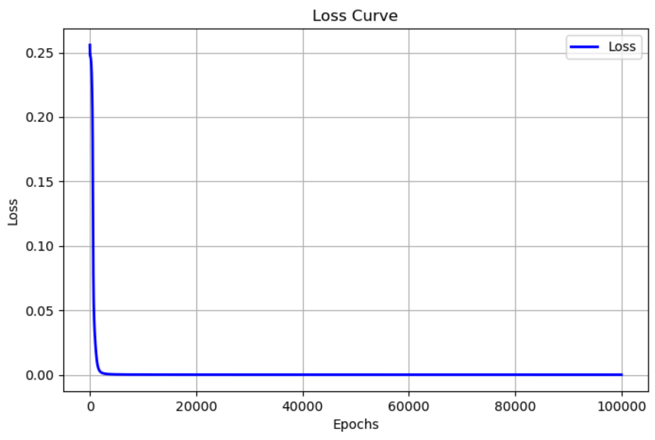
ii. XOR:

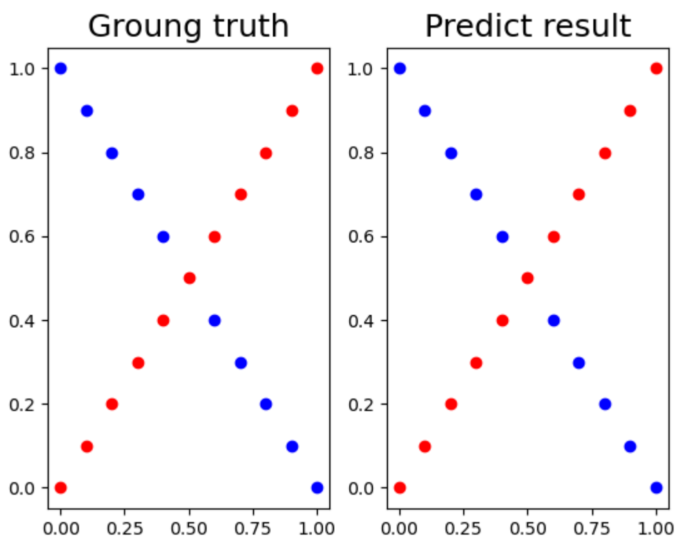
1. AdaGrad:





2. Momentum:

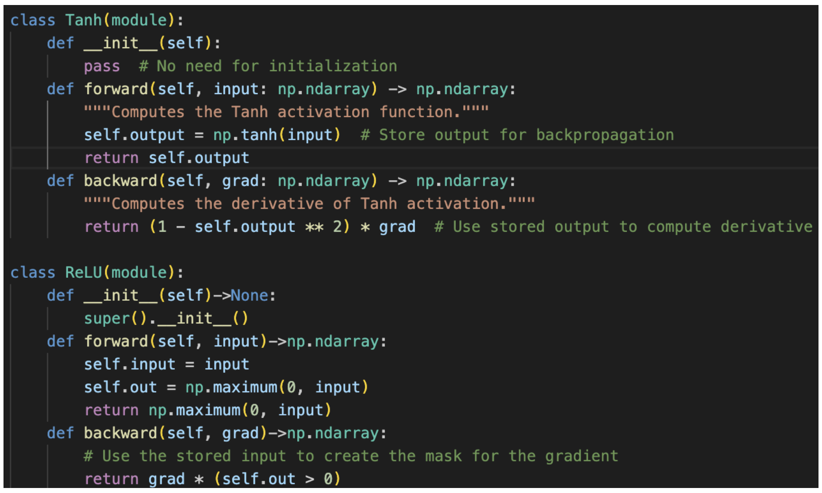




According to the observation of figures above, can discover that SGD has a more unstable convergence while adagrad and momentum has a smoother and convergence in both xor and linear dataset. However, all three optimizers have similar performance in eventual prediction.

b. Different activation functions:

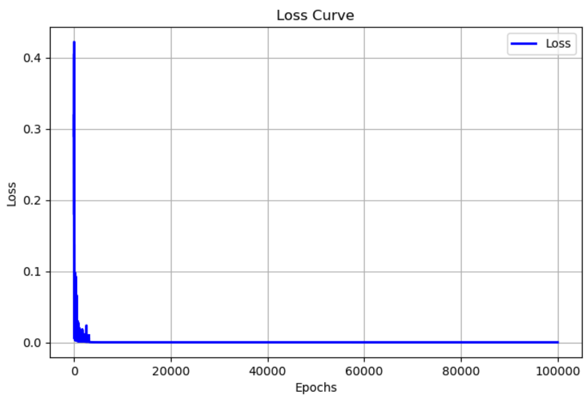
I implemented extra activation functions relu and tanh as a comparison to sigmoid. Below are comparison with network using same learning rate = 1e-6, hidden unit size = 4, sgd as optimizer but with different activation function

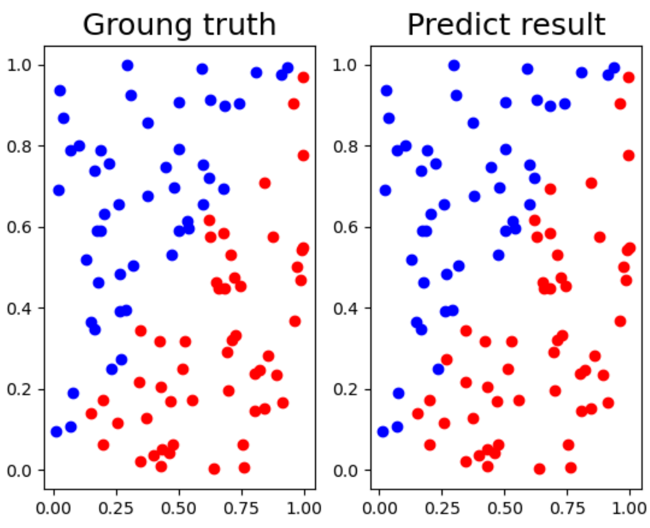


```python
class Tanh(module):
    def __init__(self):
        pass  # No need for initialization
    def forward(self, input: np.ndarray) -> np.ndarray:
        """Computes the Tanh activation function."""
        self.output = np.tanh(input)  # Store output for backpropagation
        return self.output
    def backward(self, grad: np.ndarray) -> np.ndarray:
        """Computes the derivative of Tanh activation."""
        return (1 - self.output ** 2) * grad  # Use stored output to compute derivative

class ReLU(module):
    def __init__(self)->None:
        super().__init__()
    def forward(self, input)->np.ndarray:
        self.input = input
        self.out = np.maximum(0, input)
        return np.maximum(0, input)
    def backward(self, grad)->np.ndarray:
        # Use the stored input to create the mask for the gradient
        return grad * (self.out > 0)
```

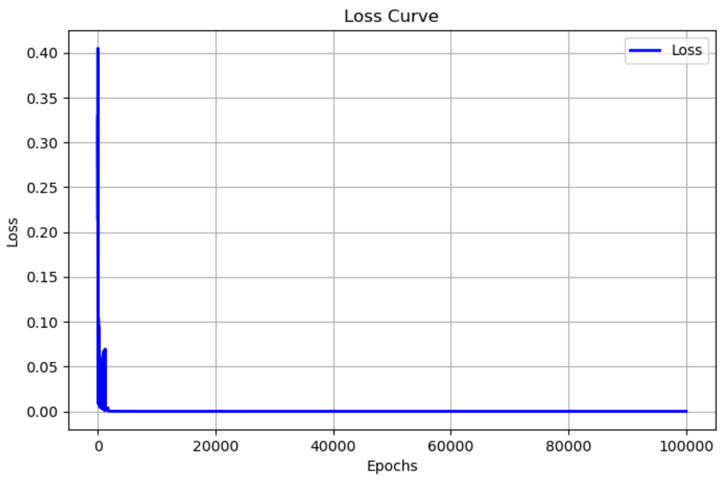
i. Linear:

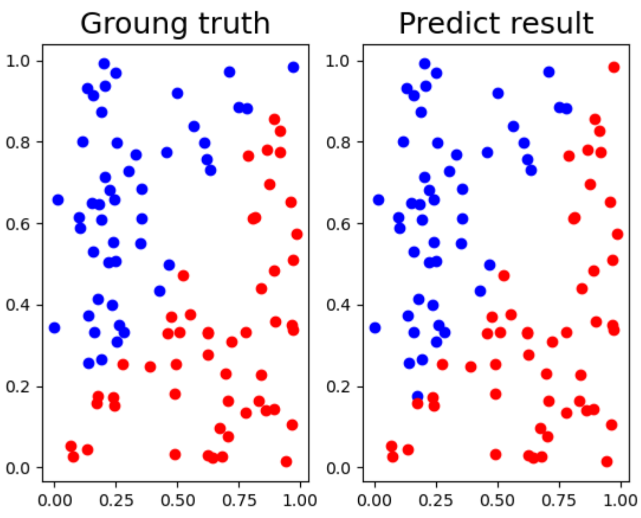
1. Relu:


Loss Curve
Loss
Loss
Epochs
0.4
0.3
0.2
0.1
0.0
0
20000
40000
60000
80000
100000


Groung truth
Predict result
1.0
0.8
0.6
0.4
0.2
0.0
0.00
0.25
0.50
0.75
1.00

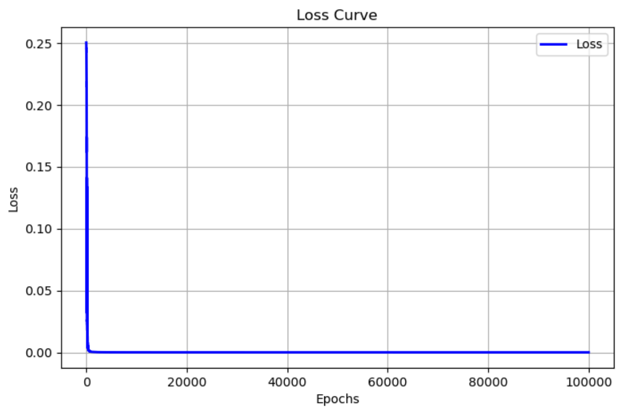
2. Tanh:

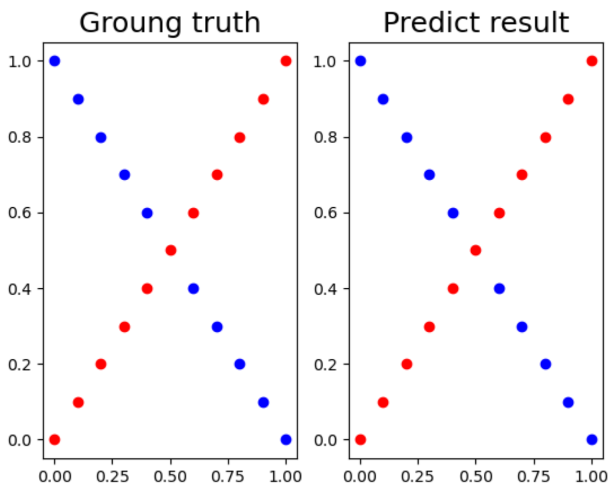

Loss Curve
Loss
Loss
Epochs
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
0
20000
40000
60000
80000
100000



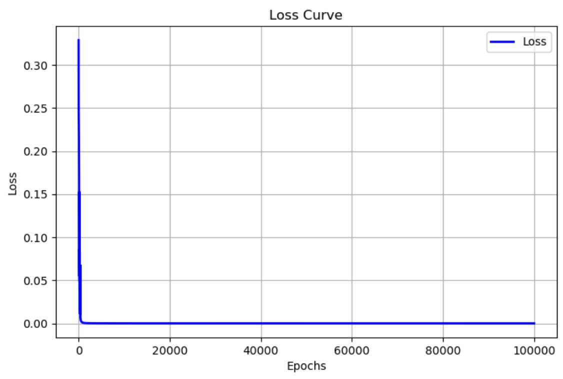
ii. **XOR:**

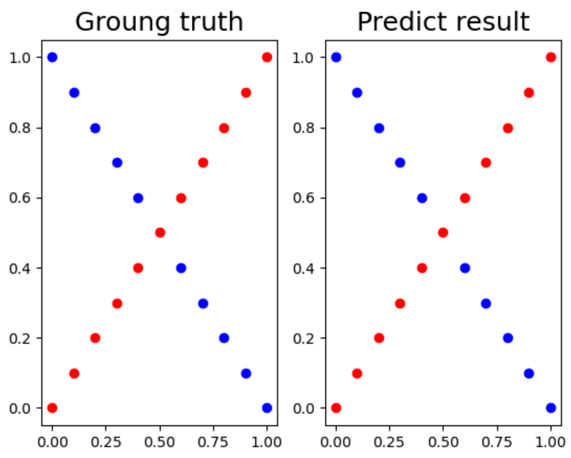
1. Relu:





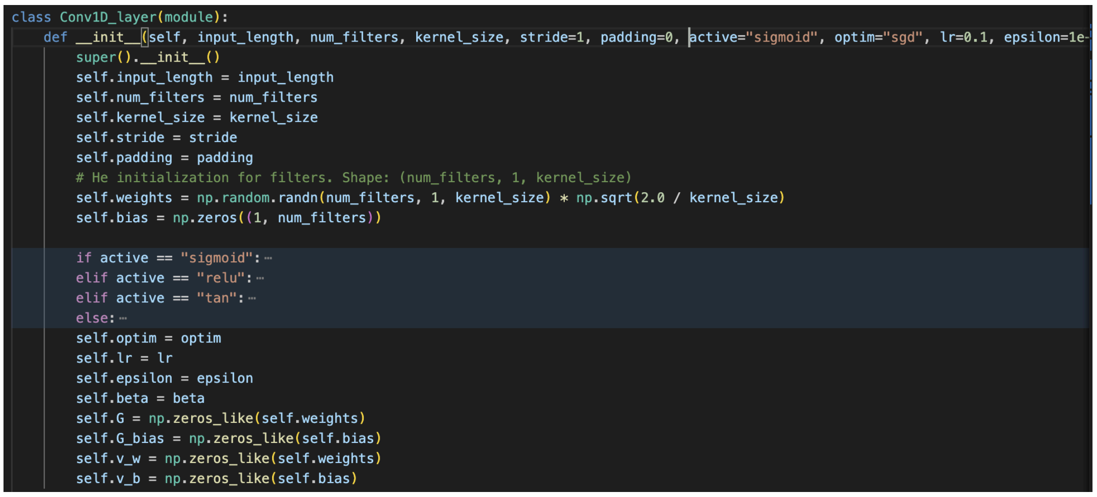
2. Tanh:



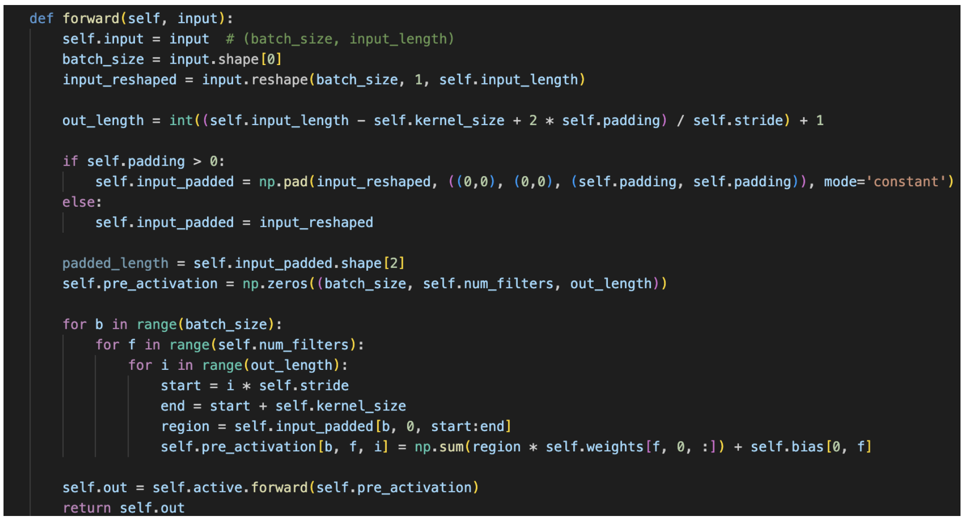


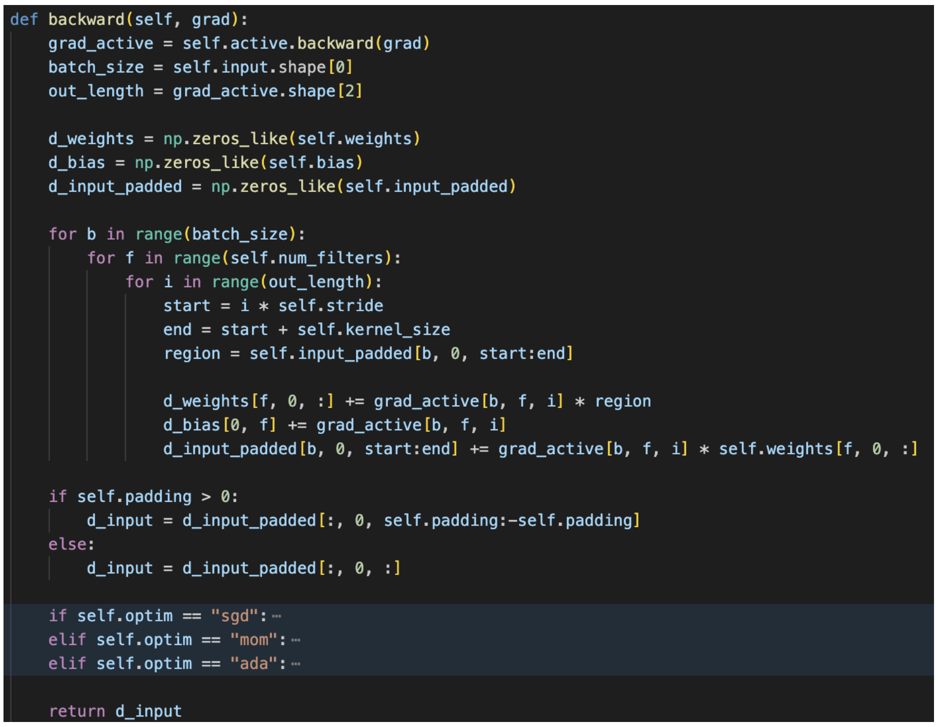
With using relu and tanh as activation function, the loss of model decrease faster and smoother compared to sigmoid in both linear and xor dataset.

c. Conv

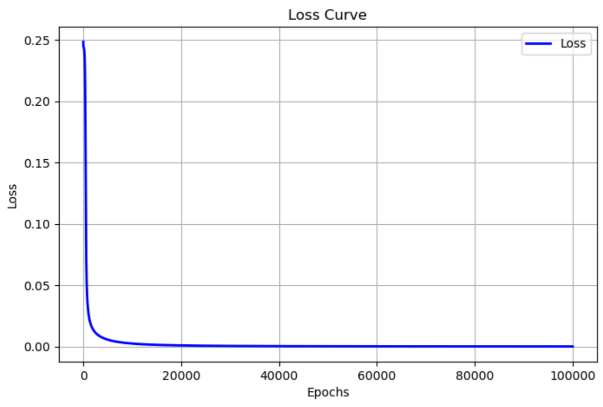


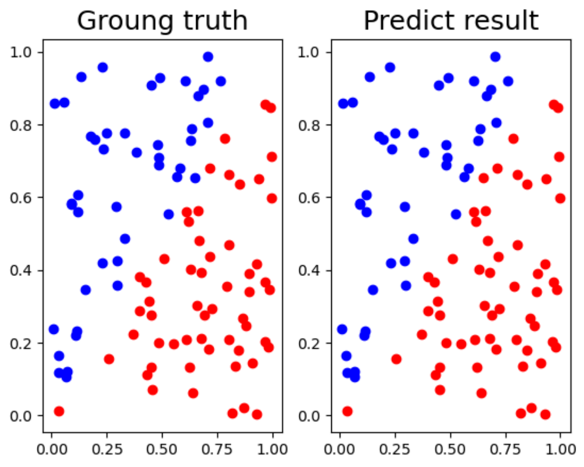
```
class Conv1D_layer(module):
    def __init__(self, input_length, num_filters, kernel_size, stride=1, padding=0, active="sigmoid", optim="sgd", lr=0.1, epsilon=1e-
        super().__init__()
        self.input_length = input_length
        self.num_filters = num_filters
        self.kernel_size = kernel_size
        self.stride = stride
        self.padding = padding
        # He initialization for filters. Shape: (num_filters, 1, kernel_size)
        self.weights = np.random.randn(num_filters, 1, kernel_size) * np.sqrt(2.0 / kernel_size)
        self.bias = np.zeros((1, num_filters))

        if active == "sigmoid":⋯
        elif active == "relu":⋯
        elif active == "tan":⋯
        else:⋯
        self.optim = optim
        self.lr = lr
        self.epsilon = epsilon
        self.beta = beta
        self.G = np.zeros_like(self.weights)
        self.G_bias = np.zeros_like(self.bias)
        self.v_w = np.zeros_like(self.weights)
        self.v_b = np.zeros_like(self.bias)
```


```python
def forward(self, input):
    self.input = input  # (batch_size, input_length)
    batch_size = input.shape[0]
    input_reshaped = input.reshape(batch_size, 1, self.input_length)

    out_length = int((self.input_length - self.kernel_size + 2 * self.padding) / self.stride) + 1

    if self.padding > 0:
        self.input_padded = np.pad(input_reshaped, ((0,0), (0,0), (self.padding, self.padding)), mode='constant')
    else:
        self.input_padded = input_reshaped

    padded_length = self.input_padded.shape[2]
    self.pre_activation = np.zeros((batch_size, self.num_filters, out_length))

    for b in range(batch_size):
        for f in range(self.num_filters):
            for i in range(out_length):
                start = i * self.stride
                end = start + self.kernel_size
                region = self.input_padded[b, 0, start:end]
                self.pre_activation[b, f, i] = np.sum(region * self.weights[f, 0, :]) + self.bias[0, f]

    self.out = self.active.forward(self.pre_activation)
    return self.out
```

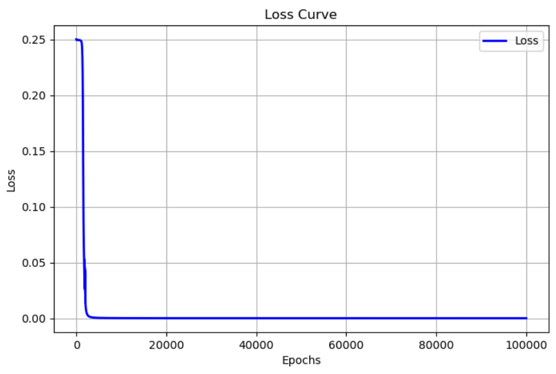

```python
def backward(self, grad):
    grad_active = self.active.backward(grad)
    batch_size = self.input.shape[0]
    out_length = grad_active.shape[2]

    d_weights = np.zeros_like(self.weights)
    d_bias = np.zeros_like(self.bias)
    d_input_padded = np.zeros_like(self.input_padded)

    for b in range(batch_size):
        for f in range(self.num_filters):
            for i in range(out_length):
                start = i * self.stride
                end = start + self.kernel_size
                region = self.input_padded[b, 0, start:end]

                d_weights[f, 0, :] += grad_active[b, f, i] * region
                d_bias[0, f] += grad_active[b, f, i]
                d_input_padded[b, 0, start:end] += grad_active[b, f, i] * self.weights[f, 0, :]

    if self.padding > 0:
        d_input = d_input_padded[:, 0, self.padding:-self.padding]
    else:
        d_input = d_input_padded[:, 0, :]

    if self.optim == "sgd":⋯
    elif self.optim == "mom":⋯
    elif self.optim == "ada":⋯

    return d_input
```

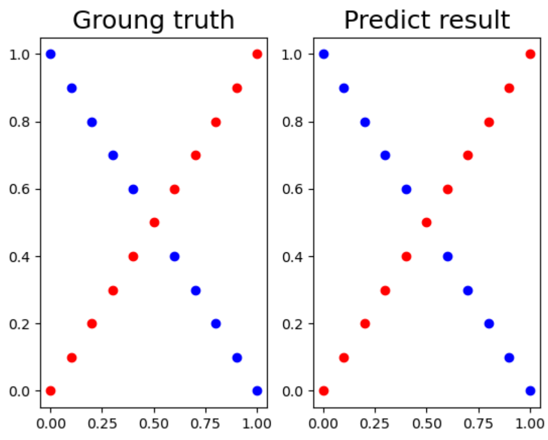
i. Linear:





ii. XOR





I implemented a model with one convolution layer followed by 2 linear layers and the outcome is shown as above. Compared to model with only linear layers, it requires more training time and larger learning rate during training session as convolution layer will capture more feature details. After multiple experiments, I discovered that with convolution layer the model has more stable accuracy. However in this lab the datasets are quite simple that can't obviously tell the difference.